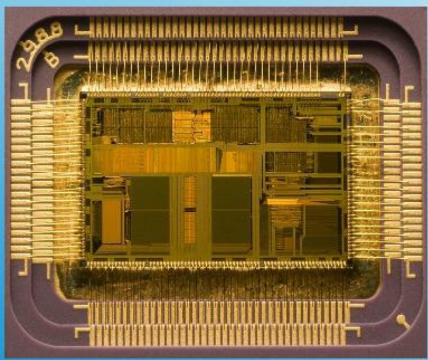
計算理論

程式人觀點



- *布林邏輯*
- *一階邏輯*
- *哥德爾定理*
- *停止問題*
- *NP-Complete*

作者：陳鍾誠 – 本書部分圖片與內容來自維基百科

前言

序

我在念碩士班的時候，修了一次「計算理論」這門課，然後在博士班的時候又修了一次，兩次都是必修課。

但是、我這兩次的課程都在似懂非懂之間就修完了。

為甚麼呢？ 我總是感到疑惑？ 計算理論中的圖靈機 (Turing Machine) 和現代電腦真的差好多，而「哥德爾」的那些奇怪的編碼方式更是和「二進位」表示法南轅北轍。

後來、我試圖用自己的語言來說明「計算理論」到底是甚麼？ 於是就寫出了您現在所看到的這本書。

希望透過我這個「現代程式人」的語言，能讓您更容易理解那些「古代學術巨人」的想法。

陳鍾誠 2014/8/13 於 金門大學 資訊工程系

授權聲明

計算理論簡介

何謂計算理論？

計算理論是資訊科學的理論基礎，主要探討電腦能力極限的問題，哪些是電腦有可能解決的問題，哪些是電腦無法解決的問題，以下是計算理論的兩大問題：

- 哪些問題是可計算的？ (What can be computed?)
- 計算該問題需要花費多少時間與空間？ (Given a problem, how much resource do we need to compute it ?)

計算理論有一些子領域，像是自動推論領域，探討的就是如何利用電腦證明數學定理。 (Prove mathematical theorem using computer) ， 但是這個問題其實不像我們想像的那麼狹窄，廣義的來看，自動推論問題其實就是在探究如何利用電腦解決問題 (Computer-based problem solving)。

舉例而言，以下是一些邏輯規則，

$$\forall x \forall y \forall z \;\; x^*(y^*z) = (x^*y)^*z$$

$$\exists x \exists y \;\; x^*y = y$$

說明：以上形式中的前面部份，是一階邏輯中的量詞限制條件，而內容部份的寫法則通常可以用下列形式表達。

$$A_1 \& ... \& A_n => P_1 \& ... \& P_n$$

在自動推論中，經常使用某些公理系統，作為推論的基本法則，這些公理系統必須具備「一致性」(consistent)，不能有邏輯矛盾的情況出現。計算機科學家必須研究如何利用這些公理証明所有可證明的定理，這幾乎就是在研究電腦能力的極限了。

邏輯推論系統

為了討論電腦的能力極限，我們往往需要藉助邏輯系統來進行描述，以下是經典邏輯系統所探討的主題，也是很多計算理論書籍的切入點。

- 邏輯系統：布林邏輯 (Boolean Logic)、謂詞邏輯 (Predicate Logic)、一階邏輯 (First Order Logic)、哥德爾完備定律、哥德爾不完備定律。

為了讓讀者感受到「邏輯與計算理論之間的關係」，請讀者先看看以下這個「皮諾公設系統」 (Peano Axiom of Natural Number System)，這是一個「數論」領域的簡單公理系統。

```
PE1 : 0 exist
PE2 : x’ = x+1
```

```
PE3 : x’ > x
PE4 : if x’ = y’ then x = y
PE5 : (數學歸納法)  if P(0) and P(x) => P(x’) then  ![](../timg/acb87fba7ea1.jpg)
```

您可以看到上述的的公理系統都是採用邏輯的方式描述的，我們可以透過數學思考去解析這類的公理系統，以變理解該「數學系統的能力極限」，而這也正是計算理論課程所想要探討的主題。

計算理論與演算法所探討的，可以說是一體兩面的東西。演算法探討用電腦解決問題的方法，但計算理論則注重電腦是否能解決該問題，或者能否在有限的時間內解決某問題。

以計算理論的角度看來，演算法所做的事情是：「尋找一個程式，該程式可以正確的輸出某個問題的答案」。如果我們將該問題改寫成邏輯數學式，則可以寫成如下的語句。

$$\forall x \;\; p(x) => \exists z \;\; q(x,z)$$

上述語句中的 p(x) 是「輸入限制函數」，而 q(x,z) 則是「輸出限制函數」。

哪些問題是可計算的？

在還沒有電腦的時代，哥德爾、圖靈等數學家就已經用數學在討論電腦能力的極限了，

- 電腦能力極限：圖靈機（Turing Machine)、停止問題 (Halting Problem)、可計算性問題。

哪些問題要算很久？

- 演算法複雜度：Big O 複雜度，多項式複雜度，指數複雜度，NP-Complete。

計算理論的經典問題

圖靈等數學家透過這些辯證探討了以下的重要主題，這些主題構成了電腦計算能力的理論核心：

- 對角證法
- 實數的數量為不可數無限大
- 羅素悖論 -- 理髮師悖論
- 有一個理髮師，他宣稱要為所有不自己剪頭髮的人剪髮，但是不為任何自己剪髮的人剪髮。
- 停止問題：請寫一個程式判斷另一個程式會不會停。
- NP-Complete：加上 Oracle (神諭) 的電腦可以在多項式時間內解決的問題。
- 多項式時間： O(1), O(n), O(n^2), O(n^3), ... , O(n^k)

- 指數時間： O(2^n), O(3^n), ... , O(k^n)

相關資源

- YouTube: 課堂錄影 -- 計算理論

邏輯世界的歷史

簡介

邏輯學是西方科學中淵遠流長的一門學問，從西元前 350 年亞里斯多德的三段論開始，就開啟了歐洲文明對邏輯學的興趣之窗。然而這一個興趣同樣隨著西方文明的發展而起伏不定，直到西元 1850 年左右，George Boole (布爾) 開始研究布林代數，才讓邏輯學成為近代數學的一個重要領域。接著，Gottlob Frege 在 1870 年左右所提出的一階邏輯系統，繼承布林系統並向上延伸，形成一個數學基礎穩固且強大的邏輯系統，於是整個經典的邏輯系統建立完成。

雖然如此，這些邏輯系統仍然是掌上的玩物，而且沒有人能確定這樣的邏輯系統，其能力到底有多強，是否一致且完備，是否有某些極限。希爾伯特在 1900 年所提出的 25 個數學問題中，這個問題被排在第二個提出。然而，希爾伯特並沒有能證明一階邏輯系統的完備性，而是在 1929 年由哥德爾證明完成了。

哥德爾的成就不僅於此，1931 年他更進一步證明了一個非常令人驚訝的定理，在「一階邏輯的擴充系統 - 皮諾數論系統」當中，不具有完備性，而且它證明了假如該系統是完備的，將會導致矛盾。

哥德爾在證明完備定理與不完備定理時，採用的都是矛盾証法，也就是透過排中律所證明的，這樣的証明並非建構性的，因此即使建立了完備定理，也沒有人能構造出一個建構式的証明方法，可以檢證一階邏輯的定理。

1965 年，Robinson 提出了一條非常簡單的邏輯證明規則 -- Resolution，並且說明了如何利用矛盾檢證程序 Refutation，證明邏輯規則在某系統中的真假，這個方法既簡單又優美，因此廣為數學界與計算機科學界所稱道。以下，我們將更詳細的說明上述人物在邏輯學上的貢獻。

亞里斯多德 (Aristotle)（出生於西元前 322 年）

亞里斯多德在其其理則學 (zoology) 研究中，提出了下列的三段式推論規則 Barbara，簡稱為三段論。

類型	語句	說明
大前提	所有人都終會死亡	普遍原理
小前提	蘇格拉底是人	特殊陳述
結論	蘇格拉底終會死亡	推論結果

布爾（Boole）（出生於 1815年）

Boole 研究邏輯時，提出了一種只有真值與假值的邏輯，稱為二值邏輯，通常我們用 0 代表假值，1 代表真值。布爾研究這種邏輯系統，並寫出了一些代數規則，稱為布林代數，以下是其中的一些代數規則。

規則 (數學寫法)	名稱
$x \vee (y \vee z) = (x \vee y) \vee z$	OR 的結合律
$x \vee y = y \vee x$	OR 的交換律
$x \wedge (y \wedge z) = (x \wedge y) \wedge z$	AND 的結合律
$x \wedge y = y \wedge x$	AND 的交換律
$\overline{x \vee y} = \overline{x} \wedge \overline{y}$	狄摩根定律(1)
$\overline{x \wedge y} = \overline{x} \vee \overline{y}$	狄摩根定律(2)

說明：上述規則中的 $\wedge$ 代表邏輯或 (AND) (在程式語言裏常寫為 & 或 and)， $\vee$ 代表邏輯或 (OR) (在程式語言裏常寫為 | 或 or)。所以若改用程式領域的寫法，可改寫如下。

規則 (數學寫法)	名稱
x | (y | z) = (x | y) | z	OR 的結合律
x | y = y | z	OR 的交換律
x & (y & z) = (x & y) & z	AND 的結合律
x & y = y & x	AND 的交換律
-(x|y) = -x & -y	狄摩根定律(1)
-(x&y) = -x | -y	狄摩根定律(2)

福雷格（Frege）（出生於 1848年）

Frege 在研究邏輯系統時，將函數的概念引入到邏輯系統當中，這種函數被稱為謂詞，因此該邏輯系統被稱為謂詞邏輯。然後，Frege 又引入了兩個量詞運算， $\forall$ (對於所有) 與 $\exists$ (存在)，透過謂詞的限定作用，以及這兩個量詞，Frege 架構出了這種具有函數的邏輯系統，後來被稱為一階邏輯系統 (First Order Logic)。

以下是我們將亞里斯多德的三段論，轉化為一階邏輯後，所寫出的一階邏輯規則。

類型	語句	說明
大前提	$\forall x \ people(x) \rightarrow mortal(x)$	所有人都終會死亡
小前提	$people(Socrates)$	蘇格拉底是人
結論	$mortal(Socrates)$	蘇格拉底終會死亡

希爾伯特 (David Hilbert)(出生於 1862年)

事實上，在電腦被發明之前，數學界早已開始探索「公理系統」的能力極限。在西元 1900 年時，德國的偉大數學家希爾伯特 (Hilbert)，提出了著名的 23 個數學問題，其中的第二個問題如下所示。

> 證明算術公理系統的無矛盾性 The compatibility of the arithmetical axioms.

在上述問題中，希爾伯特的意思是要如何證明算術公理系統的 Compatibility，Compatibility 這個詞意謂著必須具有「一致性」 (Consistency) 與「完備性」(Completeness)。

所謂的「一致性」，是指公理系統本身不會具有矛盾的現象。假如我們用 A 代表該公理系統，那麼 A 具有一致性就是 A 不可能導出兩個矛盾的結論，也就是 A => P 與 A=> -P 不可能同時成立。

所謂的「完備性」，是指所有「永遠為真的算式」(也就是定理) 都是可以被証明的，沒有任何一個定理可以逃出該公理系統的掌握範圍。

然而，希爾伯特耗盡了整個後半生，卻也無法證明整數公理系統的一致性與完備性。或許是造化弄人，這個任務竟然被希爾伯特的一位優秀學生 - 哥德爾 (Godel) 所解決了，或者應該說是否決了。

哥德爾 (Kurt Gödel)(出生於 1906 年)

哥德爾實際上證明了兩個定理，第一個是 1929 年提出的「哥德爾完備定理」(Gödel's Complete Theorem)，第二個是 1931 年證明的「哥德爾不完備定理」(Gödel's Incomplete Theorem)，這兩個定理看來似乎相當矛盾，但事實上不然，因為兩者所討論的是不同的公理系統，前者的焦點是「一階邏輯系統」(First Order Logic)，而後者的焦點則是「具備整數運算體系的一階邏輯系統」。

哥德爾完備定理證明了下列數學陳述：

一階邏輯系統是一致且完備的

一致性代表一階邏輯系統不會具有矛盾的情況，而完備性則說明了一階邏輯當中的所有算式都可以被証明或否証。

哥德爾不完備定理證明了下列數學陳述：

任何一致且完備的「數學形式化系統」中，只要它強到足以蘊涵「皮亞諾算術公理」，就可以在其中構造在體系內「既不能證明也不能否證的命題」。

哥德爾不完備定理改用另一個說法，如下所示：

如果一個包含算術的公理系統可以用來描述它自身時，那麼它要麼是不完備的，要麼是不一致的，不可能兩者皆有！

(筆者註：若該公理系統包含無限條公理時，必須是可列舉的 recursive enumerable)

羅賓遜 (John Alan Robinson) (出生於 1928 年)

雖然哥德爾證明了一階邏輯是完備的，但是卻沒有給出一個建構式的方法，可以推理出所有的的一階邏輯定理。這個問題由 John Alan Robinson 在 1965 年解決了。

Robinson 提出的 refutation 邏輯推論法是一種反證法，任何一階邏輯的算式 P 只要在系統 S 當中是真的，只要將 -P 加入該系統 S 中，就可以經由反證法導出矛盾。如果 P 在系統 S 當中不是真的，那麼將 P 加入 S 當中就無法導出矛盾。

所謂的 refutation 反證法是依靠一個稱為 resolution 的邏輯規則，該規則如下所示：

$$\frac{a_1|\ldots|a_i|\ldots|a_n \quad ; \quad b_1|\ldots|-a_i|\ldots|b_m}{a_1|\ldots|a_{i-1}|a_{i+1}|\ldots|a_n|b_1|\ldots|b_{j-1}|b_{j+1}|\ldots|b_m}$$

假如我們將上述算式中的 $a_1|\ldots|a_{i-1}|a_{i+1}|\ldots|a_n$ 寫為 A，將 $b_1|\ldots|b_{j-1}|b_{j+1}\ldots|b_m$ 寫為 B，則上述算式可以改寫如下：

$$\frac{A|a_i \quad ; \quad B|-a_i}{A|B}$$

結語

邏輯學在西方文化中扮演了非常重要的角色，而且可以說是「現代科學」會出現在歐洲的重要原因，假如將「邏輯學」從西方文化中拿掉，或許工業革命就不會出現在歐洲了？

您可以想像「孔子」整天追根究柢，常常和人辯論一件事情到底是真的還假，而且要轉換成符號，並且用邏輯的方式去證明嗎？

但是「亞里斯多德」在那個年代就是這樣追根究柢的，所以他才會去研究解剖學，把動物給切開看看裡面有甚麼，我想這也是他提出三段論背後的原因吧！

參考文獻

- 維基百科：亞里斯多德
- 維基百科：喬治·布爾
- 維基百科：三段論
- 維基百科：哥德爾不完備定理
- 維基百科：哥德爾完全性定理
- 維基百科：戈特洛布·弗雷格
- 維基百科：大衛·希爾伯特
- 維基百科：希爾伯特的23個問題
- 維基百科：庫爾特·哥德爾
- Wikipedia:Zoology
- Wikipedia:Aristotle
- Wikipedia:Boolean Logic
- Wikipedia:George Boole
- Wikipedia:Frege
- Wikipedia:Hilbert's_problems
- Wikipedia:John Alan Robinson
- Wikipedia:Resolution (Logic)
- Hilbert's Mathematical Problems
- Wikipedia:Kurt_Gödel

布林邏輯與推論系統 -- 何謂嚴格的數學證明？

前言

當我還是個學生時，我總是困惑著如何應付老師的考試，其中一個重要的數學困擾是，老師要我們「證明」某個運算式。

最大的問題不在於我不會「證明」，因為在很多科目的證明題當中，我也都「答對了」，但是這種答對總是讓我感到極度的沒有把握，因為有時老師說「這樣的證明是對的」，但有時卻說「這樣的證明是錯的」。

更神奇的是，老師的證明永遠都是對的，他們可以突然加入一個「推論」，而這個推論的根據好像之前沒有出現過，然後他們說：「由此可證」、「同理可證」....。

直到有一天，我終於懂了。

因為課堂上老師的證明往往不是「嚴格的證明」，因為嚴格的證明通常「非常的困難」，每個證明都可以是一篇論文，甚至在很多論文當中的證明也都不是嚴格的。

所以在課堂上，老師總是可以天外飛來一筆的，跳過了某些「無聊的步驟」，奇蹟式的證明了某些定理，而這正是我所以感到困擾的原因。

一般的證明

一般而言，日常生活中的證明，通常是不嚴格的。

舉例來說，我可以「證明」某人殺了死者，因為殺死死者的兇刀上有「某人」的指紋。

但是這樣的證明並不嚴格，因為有很少的可能性是「某人摸過兇刀、但是並沒有殺人」。

所以我們總是可以看到那個「外表看似小孩，智慧卻過於常人」的「名偵探柯南」，總是天外飛來一筆的「證明」了某人是兇手，這種證明與數學證明可是完全不同的。

嚴格的證明

數學的證明通常不能是「機率式」的，例如：「我證明他 99% 殺了人」，這樣的證明稱不上是嚴格的證明。

嚴格的證明也並非結果一定要是 100% 的正確 (當然也不是說結果不正確)，真正的證明是一種過程，而不是結果。

怎麼說呢？

數學其實很像程式領域的演算法，或者就像是電腦的運作過程，當我們設計出一顆 CPU 之後，你必須用該 CPU 的指令撰寫出某些函數，以便完成某個程式。

那麼，數學的 CPU 是甚麼呢？

答案是「公理系統」(Axioms)！

只有透過公理系統，經由某種演算方式，計算出待證明定理在任何情況下都是真的，這樣才算是證明了該定理。

這些公理系統其實就是數學的 CPU 指令集。

布林代數大概是數學當中最簡單的系統了，因為布林代數的值只有兩種--「真與假」(或者用 0 與 1 代表)。

為了說明嚴格的數學證明是如何進行的，我們將從布林代數的公理系統 (CPU?) 開始，說明如何證明布林代數的某些定理，就好像是如何用指令集撰寫程式一樣。

布林邏輯

對於單一變數 x 的布林系統而言，x 只有兩個可能的值 (0 或 1)。

對於兩個變數 x, y 的布林系統而言，(x, y) 的組合則可能有 (0,0), (0,1), (1,0), (1,1) 四種。

對於三個變數 x, y, z 的布林系統而言，(x, y, z) 的組合則可能有 (0,0, 0), (0,0,1), (0,1,0), (0,1,1), (1,0, 0), (1,0,1), (1,1,0), (1,1,1) 八種。

基本的布林邏輯運算有三種，AND (且), OR (或), NOT (反)，在布林代數當中，通常我們在符號上面加一個上標橫線代表 NOT，用 $\wedge$ 代表 AND，用 $\vee$ 代表 OR。

但是在程式裏面，受到 C 語言的影響，很多語言用驚嘆號！代表 NOT，用 & 代表 AND，用 | 代表 OR。以下我們將採用類似 C 語言的程式型寫法進行說明。

<table>
<tr><th>NOT</th><th>AND</th><th>OR</th></tr>
<tr>
<td><table><tr><th>x</th><th>!x</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table></td>
<td><table><tr><th>x</th><th>y</th><th>x&y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table></td>
<td><table><tr><th>x</th><th>y</th><th>x|y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table></td>
</tr>
</table>

假如我們想知到某個邏輯式的真值表，例如 (-x | y) 的真值表，只要透過列舉的程序就可以檢查完畢。

x	y	-x	-x|y
0	0	1	1
0	1	1	1
1	0	0	0
1	1	0	1

接著，我們就可以定義一些公理系統，這些「公理系統」就像是數學推理的指令集，讓我們可以推論出哪些邏輯式在這個公理系統下是真的 (定理)，哪些邏輯式這個公理系統下不一定是真的。

公理系統 1

舉例而言，假如我們制定了一個公理系統如下所示。

```
公理 1: -p | q
公理 2: p
```

那麼，我們就可以列出這個布林系統的真值表。

p	q	-p|q
0	0	1
0	1	1
1	0	0
1	1	1

在上述真值表中，凡是無法滿足公理系統的列，就代表該項目違反公理系統，因此在此公理系統下不是真的，可以被刪除 (不是該公理系統的一個「解答」)。

註：在邏輯的術語中，滿足該公理系統的解答，稱為一個 Model (模型)。

在上述表格中，前兩條的 x 為 0，因此不滿足公理 2，而第三條的 -p|q 為 0，不滿足公理 1，因此符合該公理系統的項目就只剩下了一個了。

p	q	-p|q
1	1	1

在這個滿足公理系統的真值表當中，我們可以看到 q 只能是 1，也就是 q 其實是個定理。

說明：在上述邏輯推論系統當中， -p|q 可以簡寫為 p → q，因此上述公理系統可以改寫如下，這樣的推論法則稱為 Modus Ponus (中文翻成「肯定前件」)。

```
公理 1: p → q
公理 2: p
------------------
結論:  q
```

公理系統 2

假如我們定義了以下的公理系統：

```
公理 1: -p | q
公理 2:  p | r
```

那麼我們可以列出真值表如下：

p	q	r	-p|q	p|r
0	0	0	1	0
0	0	1	1	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

當我們將不符合公理系統的項目拿掉之後，以上的真值表就只剩以下這些項目。

p	q	r	-p|q	p|r
0	0	1	1	1
0	1	1	1	1

1	1	0	1	1
1	1	1	1	1

此時，如果我們檢查這些項目中 q|r 的真值表，會發現 q|r 為真者其結果全部為 1 ，因此 q|r 在這個公理系統下是真理。

p	q	r	-p|q	p|r	q|r	p|q
0	0	1	1	1	1	0
0	1	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

但是如果我們檢查這些項目中 p|q 的真值表，會發現有一項為 0 ，因此 p|q 在這個公理系統下並非真理。

所以 q|r 在此公理系統下是一個定理，但 p|q 則不是定理。

說明：在上述邏輯推論系統當中， -p|q 可以簡寫為 p → q，而 p|r 則可以想成 -(-p)|r ，於是寫成 -p → r。

於是您可以觀察到當 p=1 時 q=1，當 p=0 時 r=1，而 p 只有可能是 1 或 0，於是 q 與 r 兩者至少有一個成立，這也就是推論出的定理 q|r 成立的原因了。

推論法則

現在，我們已經具備了足夠的基本知識，可以用來說明何謂嚴格的數學證明了。

假如我們將公理系統 2 中推論出 q|r 的程序，變成一條明文的規則，如下所示：

```
(-p | q) & (p | r) → (q|r)
```

那麼，我們就可以用這樣的規則進行推論，這個推理方式乃是 Robinson 所提出的，稱為 Resolution 法則。

於是我們可以根據這條規則，推論出某個邏輯公理系統下的定理。

必須注意的是，在以上的描述中，我們並沒有區分變項與常項。讓我們在此先說明一下。

在一般的邏輯系統中，通常我們用小寫代表變項，大寫代表常項。其中的變項可以設定成任意的項目，而常項則只能代表自己。

舉例而言，A, B, C, DOG, SNOOPY 等代表常項，而 x, y, z, w, ... 等則代表變項。

公理系統理可以包含變項與常項，舉例而言，假如有個公理系統如下所示。

```
A | -B
-A | C
-(-B | C) | D
```

而這整個邏輯系統的推論法則只有一條，那就是 Resolution 法則，也就是 (-p | q) & (p | r) → (q|r) 。

我們可以透過推論法則對公理系統中的公理進行綁定 (例如 p 設定為 A，q 設定為 -B ....) 與推論，得到下列結果：

```
(A | -B) & (-A | C) → (-B|C)  ; 令 p=A, q=-B, r=C ，於是可以推出 (-B|C)。
(-B|C) & (-(-B | C) | D) → D  ; 令 p=(-B|C), q=D, r=空集合，於是可以推出 D。
```

透過這樣的推論、我們就得到了以下的「事實庫」。

```
A | -B
-A | C
-(-B | C) | D
(-B | C)
D
```

如此我們就可以不需要依靠真值表，直接從公理系統開始，透過嚴格的計算程序，推論出該公理系統中的定理了。

這種證明方式，就是一種為嚴格的數學證明。

這種證明所遵循的，乃是一種 『公理 / 推論 / 定理1 / 推論 / 定理2 / ... 』 的方式，這種方式讓證明變成了一種計算過程，是可以寫成電腦程式的，這種證明方式乃是一種嚴格可計算的證明方式。

後記

大部分的數學系統，都希望能達到這樣嚴格的程度，但可惜的是，並非所有數學系統都能完全達到這樣嚴格的程度。舉例而言：歐氏幾何可以說是公理化的早期經典之作，但其中仰賴圖形直覺的證明過程仍然有很多，並非完全達到公理化。而微積分等數學的嚴格公理化也一直是數學家還在研究的問題。

但對公理化數學體系最精彩的一段歷史是，希爾伯特對公理化的問題與歌德爾不完備定理對

數學可完全公理化的反證，以下是這段歷史的簡要說明。

20 世紀的大數學家 Hilbert 曾經於 1900 年提出的 23 個數學問題中提到一個問題，就是「是否能為數學系統建立證明法則，讓數學證明可以完全被計算出來」，後來歌德爾 (Godel) 在 1926 年證明了一階邏輯的完備定理，讓大家看到了一線曙光，但歌德爾在 1929 年又提出了一個數論系統的不完備定理，證明了有些定理無法透過計算程序證明。

歌德爾的研究，後來在電腦領域，被圖靈 (Turing) 重新詮釋了一遍，圖靈證明了「停止問題」是電腦無法 100% 正確判定的問題，這也開啟了後來計算理論的研究之河。圖靈也因此而成為計算理論領域的第一人，所以 ACM 這個組織才會將電腦界的最重要獎項稱為「圖靈獎」(Turing Award)。

參考文獻

- 維基百科：命題邏輯
- 相關討論：為甚麼國中的數學證明是從「歐式幾何」開始教，而不從「布林代數」開始教呢？
    - https://www.facebook.com/ccckmit/posts/10151056707046893
- 數學中的公理化方法 (上) 吳開朗
    - http://w3.math.sinica.edu.tw/math_media/d171/17111.pdf
- 數學中的公理化方法 (下) 吳開朗
    - http://w3.math.sinica.edu.tw/math_media/d172/17203.pdf

謂詞邏輯、一階邏輯與「哥德爾完備定理」

前言

在 布林邏輯與推論系統 -- 何謂嚴格的數學證明？ 這篇文章中，我們介紹了「布林邏輯」(Boolean Logic) 這種簡單的推論系統，這種邏輯系統又稱為「命題邏輯」(Propositional Logic)。

在本文中，我們將介紹一個能力較強大的邏輯系統，稱為「一階邏輯」(First Order Logic) 系統，這是一種「謂詞邏輯」(Predicate Logic) 的實例，然後再說明這種邏輯系統中的一個重要定理，稱為「哥德爾完備定理」。

謂詞邏輯

在布林邏輯中，只有用來代表真假值的簡單變數，像是 A, B, C, X, Y, Z .... 等，所以邏輯算式看來通常如下：

- P & (P=>Q) => Q.
- A & B & C => D | E.
- -(A & B) <=> -A | -B.

這種命題邏輯裏沒有函數的概念，只有簡單的命題 (Proposition)，因此才稱為命題邏輯。

而在謂詞邏輯裏，則有「布林函數」的概念，因此其表達能力較強，例如以下是一些謂詞邏輯的範例。

- Parent(x,y) <= Father(x,y).
- Parent(John, Johnson).
- Ancestor(x,y) <= Parent(x,y).
- Ancestor(x,y) <= Ancestor(x,z) & Parent(z,y).

您可以看到在這種邏輯系統裏，有「布林變數」的概念 (像是 x, y, z 等等)，也有函數的概念，像是 Parent(), Father(), Ancestor() 等等。

一階邏輯

在上述這種謂詞邏輯系統中，如果我們加上 $\forall$ (對於所有) 或 $\exists$ (存在) 這兩個變數限定符號，而其中的謂詞不可以是變項，而必須要是常項，這種邏輯就稱為一階邏輯。

- $\forall People(x) => Mortal(x)$; 人都是會死的。
- $People(Socrates)$; 蘇格拉底是人。
- $Mortal(Socrates)$; 蘇格拉底會死。

當然、規則可以更複雜，像是以下這個範例，就說明了「存在一些人可以永遠被欺騙」。

- $\exists x(Person(x) \& \forall y(Time(y) => Canfool(x,y)))$.

二階邏輯

如果一階邏輯中的謂詞，放寬成可以是變項的話 (這些變項可以加上 $\forall$ 與 $\exists$ 等符號的約束)，那就變成了二階邏輯，以下是一些二階邏輯的規則範例。

- $\exists P(P(x) \& P(y))$.
- $\forall P \forall x(x \in P | x \notin P)$.
- $\forall P(P(0) \& \forall y(P(y) => P(succ(y))) => \forall y P(y))$. ; 數學歸納法。

一致性與完備性

在邏輯系統中，所謂的「一致性」，是指公理系統本身不會具有矛盾的現象。假如我們用 A 代表該公理系統，那麼 A 具有一致性就是 A 不可能導出兩個矛盾的結論，也就是 A => P 與 A=> -P 不可能同時成立。

哥德爾完備性定理

哥德爾於 1929 年證明了「哥德爾完備定理」(Gödel's Complete Theorem)，這個定理較簡化的陳述形式如下：

- 一階邏輯系統是一致且完備的，也就是所有的一階邏輯定理都可以透過機械性的推論程序證明出來，而且不會導出矛盾的結論。

以下是哥德爾完備定理的兩種陳述形式，詳細的證明方法請參考 Wikipedia:Original proof of Gödel's completeness theorem。

- Theorem 1. Every formula valid in all structures is provable.
- Theorem 2. Every formula φ is either refutable or satisfiable in some structure

結語

「哥德爾完備性定理」似乎得到了一個很正向的結果，讓人對邏輯系統的能力擁有了一定的信心。

但是、當哥德爾進一步擴展這個邏輯系統，加入了「自然數的加法與乘法」等運算之後，卻發現了一個令人沮喪的結果，那就是「包含自然數加法與乘法的一階邏輯系統，如果不是不一致的，那就肯定是不完備的，不可能兩者都成立」。

這將引出我們的下一篇文章， 從程式人的角度證明「哥德爾不完備定理」。

參考文獻

- 維基百科：謂詞邏輯
- 維基百科：一階邏輯
- 維基百科：二階邏輯
- Wikipedia:First-order logic
- Wikipedia:Second-order_logic
- 維基百科：哥德爾完備性定理
- http://www.encyclopediaofmath.org/index.php/Henkin_construction
- Wikipedia:Original proof of Gödel's completeness theorem

從程式人的角度證明「哥德爾不完備定理」

1900 年，德國的偉大數學家希爾伯特 (Hilbert)，提出了著名的 23 個數學問題，其中的第二個問題如下所示。

> 證明算術公理系統的無矛盾性 The compatibility of the arithmetical axioms.

在上述問題中，希爾伯特的意思是要如何證明算術公理系統的 Compatibility，Compatibility 這個詞意謂著必須具有「一致性」 (Consistency) 與「完備性」(Completeness)。

為此、許多數學家花費了一輩子的心力，企圖建構出一個「既一致又完備」的邏輯推論系統，像是「羅素與懷德海」就寫了一本「數學原理」，希望為數學建構出非常扎實的「公理系統」。

結果、這樣的企圖心被哥德爾的一個定理給毀了，那個定理就是「哥德爾不完備定理」。

要瞭解「哥德爾不完備定理」之前，最好先瞭解一下「邏輯悖論」這個概念。

當初、羅素在努力的建構數學原理時，卻發現了數學中存在著邏輯悖論，於是發出感嘆：「當我所建構的科學大廈即將完工之時，卻發現它的地基已經動搖了...」。

羅素的話，其原文是德文，據說翻譯成英文之後意義如下：

> Hardly anything more unwelcome can befall a scientific writer than that one of the foundations of his edifice be shaken after the work is finished

結果，在 1950年，羅素穫得諾貝爾文學獎 (天啊！羅素不是數學家嗎！但是看他上面那句話的文筆，我很能體會他得諾貝爾文學獎的原因了 ...)

理髮師悖論

理髮師悖論可以描述如下：

> 在某一個小世界裏，有一個理髮師，他宣稱要為該世界中所有不自己理頭髮的人理髮，但是不為任何一個自己理頭髮的人理髮！
>
> 請問、他做得到嗎？

您覺得呢？

這個問題的答案是，他絕對做不到，原因出在他自己身上：

> 如果他「為」自己理頭髮，那麼他就為「一個自己理頭髮的人理髮」，違反了後面的宣言。
>
> 如果他「不為」自己理頭髮，那麼他就沒有為「該世界中 "所有" 不自己理頭髮的人理髮」，因此違反了前面的宣言。

於是、他理也不是、不理也不是，這就像中國傳說故事裏「矛與盾」的故事一樣，他的問題陷入兩難，產生「矛盾」了。

所以、該理髮師想做的事情是不可能做得到的！

這樣的悖論，在邏輯與電腦的理論裏有很深遠的影響，哥德爾正是因為找到了邏輯體系的悖論而發展出「哥德爾不完備定理」，而電腦之父圖靈也事發現了「停止問題」會造成悖論而證明了有些事情電腦做不到 ....

哥德爾不完備定理的描述

當初「哥德爾」提出的「不完備定理」，大致有下列兩種描述方法，後來簡稱為「哥德爾第一不完備定理」與「哥德爾第二不完備定理」，如下所示。

哥德爾第一不完備定理

> 定理 G1：若公理化邏輯系統 T 是個包含基本算術 (皮諾公設)的一致性系統，那麼 T 中存在一種語句 S，但是你無法用 T 證明 S ，卻也無法否證 S。

哥德爾第二不完備定理

> 定理 G2：若公理化邏輯系統 T 是個包含基本算術 (皮諾公設)的一致性系統，那麼 T 無法證明自己的一致性。

但是、對於「程式人」而言，上述描述都太邏輯了，讓我們改用「程式人」的角度來看這個問題，提出另一種「程式型版本」的說法：

哥德爾不完備定理的程式型：

> 定理 G3：不存在一個程式，可以正確判斷一個「包含算術的一階邏輯字串」是否為定理。

哥德爾不完備定理的程式型證明

接著、就讓我們來「證明」一下上述的程式型「哥德爾不完備定理」吧！

由於牽涉到矛盾，所以我們將採用反證法：

證明：

假如這樣一個程式存在，那麼代表我們可以寫出一個具有下列功能的函數。

```
function Proveable(str)
  if (str is a theorem)
    return 1;
  else
    return 0;
end
```

這樣的函數本身，並不會造成甚麼問題，「包含算術的一階邏輯」(簡稱為 AFOL) 夠強，強到可以用邏輯式描述 Provable(str) 這件事，因此我們可以寫出 Provable(s) 這樣一個邏輯陳述。

更厲害的是，我們也可以將一個字串在 AFOL 裏，是否為定理這件事情，寫成邏輯陳述 (註：邏輯符號 ∃ 代表存在，- 代表 not， & 代表 and，| 代表 or)。

接著、我們就可以問一個奇怪的問題了！那個問題描述如下。

> 請問 isTheorem(∃s -Provable(s) & -Provable(-s)) 是否為真呢？

讓我們先用 T 代表 ∃s -Provable(s) & -Provable(-s) 這個邏輯式的字串，然後分別討論「真假」這兩個情況：

1. 如果 isTheorem(T) 為真，那麼代表存在無法證明的定理，也就是 Provable 函數沒辦法證明所有的定理。

2. 如果 isTheorem(T) 為假，那麼代表 -T 應該為真。這樣的話，請問 Provable(-T) 會傳回甚麼呢？讓我們分析看看：

```
function Proveable(-T)
  if (-T is a theorem) // 2.1 這代表 -(∃s -Provable(s) & -Provable(-s)) 是個定理，也就是 Provable() 可以正確證明所有定理。
    return 1;          //      但這樣的話，就違反了上述 「2. 如果 isTheorem(T) 為假」的條件了。
  else                 // 2.2 否則代表 -T 不是個定理，也就是存在 (∃) 某些定理 s 是無法證明的。
    return 0;          //      但這樣的話，又違反上述「2. 如果 isTheorem(T
```

```
) 為假」的條件了。
end
```

於是我們斷定：如果 Provable() 對所有輸入都判斷正確的話，那麼 2 便是不可能的，因為 (2.1, 2.2) 這兩條路都違反 2 的假設，也就是只有 1 是可能的，所以我們可以斷定 Provable(s) 沒辦法正確證明所有定理。

結語

在本文中，我們沒有寫出 Provable(s) 的邏輯陳述，也沒有寫出 isTheorem() 的邏輯陳述，因為這需要對「程式的指令集」，也就是 CPU 做一個邏輯描述，這樣說來故事就太長了！

而這個 CPU，通常後來的「計算理論」書籍裏會用「圖靈機」來描述，但這並不是哥德爾當初的證明，因為「哥德爾證明不完備定理」的年代，圖靈還沒有提出「圖靈機」的概念。

事實上、當初「哥德爾」的證明，根本也沒有「程式與電腦的概念」，所以「哥德爾」花了很多力氣建構了一個「哥德爾化的字串編碼概念」，這種字串編碼是建構在包含「+, *」兩個運算的算術系統上，也就是「皮亞諾公設」所描述的那種系統。這也是為何要引進「算術」到一階邏輯中，才能證明「哥德爾不完備定理」的原因了。

1931 年「哥德爾」證明出「不完備定理」之後，後來「圖靈」於 1936 年又提出了一個電腦絕對無法完全做到的「停止問題」(Halting Problem)，該問題乃是希望設計出一個函數 isHalting(code, data) ，可以判斷程式 code 在輸入 data 之後會不會停，也就是 code(data) 會不會停。圖靈利用圖靈機的架構，證明了該問題同樣是不可判定的，也就是沒有任何一個程式可以完全正確的判定這樣的問題。

「圖靈」的手法，與「哥德爾」非常類似，但是卻又更加簡單清楚。(不過既使如此，我還是很難直接理解圖靈的證明，因為本人在碩博士時連續被「圖靈機」荼毒了兩次，再也不希望跟「圖靈機」有任何瓜葛了 ....)

但是、我們仍然希望能夠讓「對程式有興趣」的朋友們，能夠清楚的理解「圖靈」與「哥德爾」在「計算理論」上的成就與貢獻，以免過於自大的想寫出一個「可以解決所有問題的程式」，我想只有站在前人的肩膀上，才能看清楚「程式」到底是個甚麼東西吧！

(當然、其實想要「寫出一個可以解決所有問題的程式」是非常好的想法。雖然「圖靈」與「哥德爾」已經都告訴過我們這是不可能的，但是身為一個程式人，就應該有挑戰不可能任務的決心，不是嗎？ .... 雖然、不一定要去做這種不可能的問題啦 ....)

參考文獻

- Wikipedia:Russell's paradox
- 維基百科:羅素悖論
- An Outline of the Proof of Gödel's Incompleteness Theorem, All essential ideas - without the

final technical details.

- Godel's Incompleteness Theorem, By Dale Myers
- 哥德尔轶事
- A Short Guide to Godel's Second Incomplete Theorem (PDF), Joan Bagaria.
- Wikipedia:Proof sketch for Gödel's first incompleteness theorem

停止問題

圖靈與停止問題

圖靈是計算理論的先驅，他在 1936年於「On Computable Numbers, with an Application to the Entscheidungsproblem」 這篇論文中提出了「圖靈機」的概念，並且證明了「停止問題」是任何圖靈機都無法完美解答的問題，以下是停止問題的簡單描述。

```
1. 請問您是否有辦法寫一個程式，判斷另一個程式會不會停，
2. 如果會停就輸出 1，不會停就輸出 0。
```

停止問題不可判定

由於圖靈的證明是建構在圖靈機上的，而圖靈機又很難用幾句話簡單描述，因此我們改用「現代程式」的方法證明停止問題，證明過程如下：

停止問題採用教數學的方式來說，是我們想定義一個函數 isHalt(code, data) ， 該函數可以判斷程式 code 在輸入 data 之後，是否會停止， 也就是 code(data) 會不會停止。

如我用程式寫下來，可寫成如下的演算法：

```
isHalt(code, data) = 1  假如 code(data) 會停就輸出 1
                   = 0  假如 code(data) 不停就輸出 0
```

但是、假如上述函數真的存在，那麼我們就可以寫出下列這個函數：

```
function U(code) // 故意用來為難 isHalt(code, data) 的函數。
  if (isHalt(code, code)==1) // 如果 isHalt(U, U)=1，代表判斷會停
    loop forever //   那 U 就進入無窮迴圈不停了，所以 isHalt(U,U) 判斷錯誤了。
  else             // 如果 isHalt(U, U)=0，代表判斷不停
    halt           //   那 U 就立刻停止，所以 isHalt(U,U) 又判斷錯誤了。
end
```

如此、請問 isHalt(U,U) 應該是甚麼呢？ 這可以分成兩種情況探討：

```
1. 假如 isHalt(U, U) 傳回 1，那麼就會進入無窮回圈 loop forever，
   也就是 U(U) 不會停

=> 但是 isHalt(U,U)=1 代表 isHalt 判斷 U(U) 是會停的啊?
   於是 isHalt(U, U) 判斷錯誤了。
```

```
2. 假如 isHalt(U, U) 傳回 0，那麼就會進入 else 區塊的 halt，
   也就是 U(U) 會立刻停止

=> 但是 isHalt(U,U)=0 代表 isHalt 判斷 U(U) 是不會停的啊？
   於是 isHalt(U, U) 又判斷錯誤了。

於是、我們證明了停止問題是不可能做到 100% 正確的，
因為 isHalt 永遠對 U(U) 做了錯誤的判斷。
```

停止問題的意義

從上述的論證中，我們看到 isHalt(code, data) 這個問題是無法被 100% 正確解答的，因為這個問題與「羅素的理髮師問題」一樣，都是會導制矛盾的，因此我們可以根據「矛盾證法」的推論，發現這樣的問題是無法被「圖靈機或現代電腦」所正確解答的。

在「韓非子/難一」篇當中，曾經提到一個「矛與盾」的故事，原文截錄如下：

> 楚人有鬻楯與矛者，譽之曰：『吾楯之堅，物莫能陷也。』又譽其矛曰：『吾矛之利，於物無不陷也。』或曰：『以子之矛陷子之楯，何如？』其人弗能應也。

現代電腦的能力基本上也只相當於一個記憶空間有限的圖靈機，因此一但證實了圖靈機無法解決某問題，那麼現代電腦也就無法解決該問題了。

相反的、假如我們可以證明「一個擁有無限記憶體的現代電腦」無法解決某個問題，那麼、應該也就可以證明圖靈機無法解決該問題了。

NP-Complete 問題

演算法的複雜度

通常我們會用 BigO 的觀念來描述一個演算法的複雜度。舉例而言、泡沫排序 (Bubble Sort) 的複雜度為 $O(n^2)$ ，而插入排序 (Insertion Sort) 的複雜度則為 $O(nlogn)$ 。

對於那些複雜度可用 $O(n^k)$ 規範的演算法而言，在 n 很大的時候，其成長速率會比 $O(2^n)$ 這類的函數要慢上許多。因此、在理論上而言，我們寧可用 $O(n^k)$ 的演算法，也不要用 $O(2^n)$ 的演算法。

我們稱那些複雜度受 $O(n^k)$ 限制的演算法為「多項式時間演算法」 (Polynomial Time Algorithm)， 而那些超越 $O(n^k)$ 限制，但受 $O(2^n)$ 限制的演算法為「指數時間演算法」(Exponential Time Algorithm)。

但是、有些問題很明確的需要 $O(2^n)$ 的時間，例如河內塔問題，在圓盤數為 n 的時候，需要移動 2^n-1 次才能完成。

不過、有些問題我們並不知道需要多久的時間才能完成，甚至不知道其複雜度到底是 $O(n^k)$ 或 $O(2^n)$ ，在這類的問題當中，有一群稱為 NP-Complete 的問題，特別受到「計算機科學」領域的學者所重視。

非決定性演算法 (Nondeterministic algorithm)

在電腦領域，非決定性演算法是指那些「針對相同的輸入，每次執行結果可能不同的演算法」，像是「平行的演算法」就會與「執行順序」有關，而「隨機式演算法」則會與「亂數的產生方式」有關。

NP (Nondeterministic Polynomial Time) 問題

如果一個「隨機式演算法」有時只需要「多項式時間」，但有時又需要「指數時間」才能完成，這類的演算法就稱為「非決定性多項式時間」(Nondeterministic polynomial time) 演算法。

而那些可以用「非決定性多項式時間演算法」解決的問題，我們就稱為 NP 問題。

當然、有很多問題都屬於 NP 問題。

舉例而言、像是排序問題可以用 O(N log N) 複雜度的演算法解決，由於 $NlogN<N^2$ ，所以排序問題的複雜度低於 O(N^2) ，所以當然屬於「多項式時間」的問題。

同樣的、像是「搜尋、矩陣相乘、計算反矩陣、....」等等常見的問題，幾乎都屬於「多項式時間」的問題，當然也可以用「決定性多項式時間的演算法」(Deterministic Polynomial Time Algorithm)來解決。

事實上，上述那些明確受制於 $O(N^k)$ 的問題，像是「排序、搜尋、矩陣相乘、計算反矩陣、....」等等，都屬於「決定性多項式問題」(Polynomial Time Problem)，簡稱 P 問題。

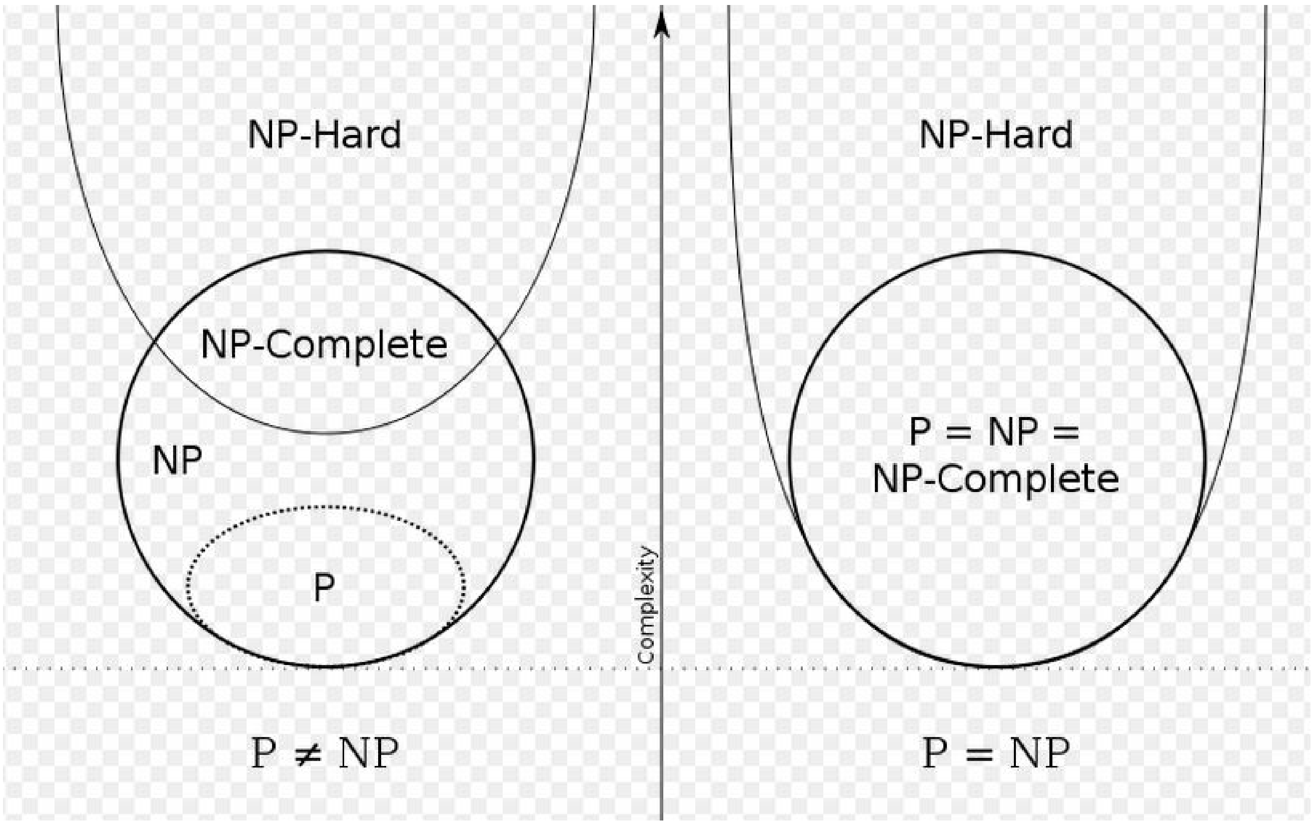
但是、對於某些問題，我們知道當採用「非決定性演算法」的時候，有時可以很快的傳回解答 (在 $O(n^k)$ 之內)，有時卻又要很久才能傳回解答 (需要 $O(2^n)$ 的步驟)，但是我們卻沒有把握每次都能多項式時間 $O(n^k)$ 內傳回解答。這種問題屬於「非決定性多項式時間」當中較難的一類問題，這類問題是我們特別想要關注的「較難的 NP 問題」。

NP-Complete 問題

在這類較難的 NP 問題當中，有一群很特別的問題，他們相互之間可以互相轉換 (在多項式時間內)，只要其中一個問題可以在多項式時間內解決，那麼其他問題也都將可以在多項式時間內解決，這一群問題稱為 NP-Complete 問題。

而那些至少與 NP-Complete 問題一樣難，甚至是更難的問題，則稱為 NP-Hard 問題。

下圖顯示了 P, NP, NP-Complete, NP-Hard 等四類問題之間的關係。



由於到目前為止，我們並不知道 NP 與 P 問題兩者是否相等，也就是無法找到一個問題是落在 NP 當中，卻又能證明不屬於 P ，因此我們無法知道 「P、NP 與 NP-Complete」之間的關係究竟應該是如上述左圖或右圖的情況。

從 1971 年 Stephen Cook 提出 NP-Complete 概念與證明以來，從來沒有人能有效解答「P 是否等於 NP」的這個問題，因此這個問題已經成為資訊領域當中最大的謎團之一。 (Stephen Cook 因此在 1982 獲得了「圖靈獎」 (Turing Award)。

後來、Richard Karp 緊接著在 「Reducibility Among Combinatorial Problems」這篇論文中提出

了 21 個互相可化約的 NP-Complete 問題。後來 Richard Karp 在 1985 年更因此獲得了「圖靈獎」。

目前已知的 NP-Complete 問題有很多，以下是一些具有代表性的 NP-Complete 問題。

- Boolean satisfiability problem (SAT) -- 布林式滿足問題
- Knapsack problem -- 背包問題
- Hamiltonian path problem -- 漢彌爾頓路徑問題
- Travelling salesman problem -- 旅行推銷員問題
- Graph coloring problem -- 著色問題

SAT 問題

在上述的 NP-Complete 問題當中，SAT 問題是特別具有歷史價值的，因為 SAT 問題是 Stephen Cook 提出 NP-Complete 概念的關鍵，所以我們必須先理解 SAT 問題，才能理解 NP-Complete 理論的核心。

SAT 問題的全稱是 Boolean satisfiability problem，也就是「布林式滿足問題」，要瞭解 SAT 問題，首先必須先瞭解何為「布林代數式」。(請參考本書 「布林邏輯」 一章)。

舉例而言，以下是一些布林代數式：

- 範例 1: (A | B | C)&(B | -A | -C)
- 範例 2: (P & -P)|(Q & -Q)
- 範例 3: (P | -P)&(Q | -Q)

針對上述的「布林代數式」，您可以看到範例 2 是無法被滿足的，因為不管 P 與 Q 如何指定，(P & -P) 和 (Q & -Q) 永遠都會傳回 false，因為兩者都是矛盾式，所以無法被滿足。

相反的、範例 3 是永遠都會被滿足的，不管我們怎麼指定， (P | -P) 永遠為真，而(Q | -Q)也是一樣，這種「布林代數式」稱為恆真式 (Tautolog，有人採用音譯的方式，翻譯成套套邏輯)。

對於範例 1 而言，如果我們指定 (A=0, B=1, C=0)，那麼該運算式將會被滿足 (傳回 1, 也就是 true)。如果我們指定 (A=0, B=0, C=0)，那麼該運算式將會傳回 0 (也就是 false)。這種可備滿足的「布林代數式」就稱為 satisfiabile。

如果任意給定一個「布林代數式」，我們是否能設計一個演算法去找出滿足該「布林式」的解答呢？這個問題就稱為 Boolean satisfiability problem，也就是 SAT 問題了。

證明：SAT 是 NP-Complete 問題

那麼、SAT 為何是 NP-Complete 問題呢？換句話說、為何所有的「非決定性多項式時間演算

法」(Nondeterministic polynomial time algorithm) 所能解決的問題 (NP 問題)，都可以化約為 SAT 問題呢？

關於這個問題，得先讓我們仔細想想到底「非決定性演算法」在做些甚麼事情？

更明確的說，一個「非決定性的圖靈機」到底在做些甚麼事情？

您只要看看下列表格，就能夠理解 Stephen Cook 到底在玩些甚麼把戲了！

如果我們用 (Q, Σ, s, F, δ) 來描述一台非決定性圖靈機，其中各個符號的意義如下：

符號	說明
Q	狀態集合
Σ	磁帶上的字母集合
s	起始狀態，是 Q 中的一個元素
F	結束狀態，是 Q 的子集合
δ	轉換關係，是 ((Q F) × Σ) × (Q × Σ × {−1, +1}) 的子集合

接著我們可以定義一大群布林變數如下：

符號	說明
T_{ijk}	當磁帶的第 i 格為符號 j (在第 k 步時) 傳回 true
H_{ik}	當機器的讀寫頭在第 i 格上 (在第 k 步時) 傳回 true
Q_{qk}	當機器處於狀態 q 時 (在第 k 步時) 傳回 true

然後、根據上述的符號定義，我們可以寫出一大堆邏輯式來描述這台「非決定性圖靈機」的行為，如下表所示：

布林運算式	條件	說明
$T_{ijk} \rightarrow -T_{ij'k}$	j ≠ j′	磁帶上每格只能有一個符號
$T_{ijk} \& T_{ij'(k+1)} \rightarrow H_{ik}$	j ≠ j'	磁帶未被寫入時符號不變
$Q_{qk} \rightarrow -Q_{q'k}$	q ≠ q'	機器在單一時間只能

		有一個狀態
$H_{ik} \rightarrow -H_{i'k}$	$i \neq i'$	讀寫頭單一時間只能有一個狀態
$(H_{ik} \& Q_{qk} \& T_{i\sigma k}) \rightarrow \bigvee_{(q,\sigma,q',\sigma',d)\in\delta} (H_{(i+d)(k+1)} \& Q_{q'(k+1)} \& T_{i\sigma'(k+1)})$	$k<p(n)$	第 k 步時讀寫頭在位置 i 的狀態轉移描述
$\bigvee_{f\in F} Q_{fp(n)}$		最後結束時必須在接受狀態

如此我們就可以完整的描述一台「非決定性圖靈機」的行為，並且將這些行為與 「SAT 布林代數式滿足問題」畫上等號，只要該圖靈機最後會停在接受狀態，「SAT 布林代數式就能被滿足」。

透過這種方式，Stephen Cook 將「非決定性圖靈機」轉換成 SAT 問題，然後證明了 「SAT 問題的滿足與 NP 問題是否有解等價」，於是 NP 問題巧妙的轉化成 SAT 問題，並且創造出了 NP-Complete 這個奇特的概念。然後當 Richard Karp 找出了 21 個可化約為其他 NP-Complete 的問題之後，這一整群的問題就通通都被視為計算複雜度上等價的問題了。

結語

雖然 Stephen Cook 所造出來的這個「SAT 布林代數式」很大，但該代數式與「非決定性圖靈機」之間的轉換卻可以在「多項式時間內」轉換完成 (幾乎都小於 $O(n^2)$)，因此這個轉換並不會造成複雜度膨脹太大的問題。

透過這種轉換方式，SAT 成了全世界第一個 NP-Complete 問題，而對於某個問題 X 而言，只要我們能在多項式時間內將 SAT 問題或任何一個已知的 NP-Complete 問題轉換為 X (在多項式時間內)，那麼就可以再度找到一個 NP-Complete 問題，於是 NP-Complete 問題就成了一個互相可化約 (reduce) 的族群。

參考文獻

- Wikipedia:NP完全
- Wikipedia:NP-complete
- Wikipedia:河內塔
- Wikipedia:Oracle machine
- Wikipedia:預言機
- Wikipedia:Cook-Levin理論
- Wikipedia:Nondeterministic algorithm
- Wikipedia:Cook–Levin theorem

結語

撰寫至此，筆者已經將「計算理論」裏較重要的問題都用自己的語言闡述了一遍，希望透過這種「較接近人類思考」的語言，讓讀者能用比較輕鬆的方式，快速瞭解那些「先知」所提出的各種奇思妙想，雖然這樣的訓練並不能讓大家寫出更強大的程式，但是卻能讓我們對電腦的理解，升華到一種異常高遠的層次。

不知哪位偉人曾經說過：「我們只有站在巨人的肩膀上、才能看得更高更遠」！

而在計算理論的世界裏，「圖靈、哥德爾、Steven Cook」等人，就是那些巨人，如果您想在計算理論上有新的突破，理解這些巨人的思想應該是有所幫助的。

筆者也是在努力理解這些「學術巨人」的過程中，撰寫出本書的，也希望這本書對讀者能有所幫助！